Computing Slices for Interprocedural Programs

A Thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology in Computer Science and Engineering

BY

Subhendu Mishra(109CS0077)

Under the Guidance of

Prof. D. P. Mohapatra

Department of Computer Science and Engineering
National Institute of Technology,
Rourkela - 769008
May - 2013

NATIONAL INSTITUTE OF TECHNOLOGY, ROURKELA

Certificate

21 May 2013

This is to certify that the thesis entitled, '**Computing Slices for Interprocedural Programs**' by **Subhendu Mishra** bearing roll number 109CS0077 in partial fulfilment of the requirements for the award of Bachelor of Technology Degree in Computer Science and Engineering at the National Institute of Technology, Rourkela is an original work carried out by him under my supervision and guidance. To the best of my knowledge the matter embodied in the thesis has not been submitted to any other University/ Institute for the award of any Degree or Diploma.

Prof D.P. Mohapatra

Acknowledgement

The project work would have been incomplete without mentioning those who have provided their help and guidance. I would like to express my gratitude to Prof D.P. Mohapatra for his constant advice and guidance throughout my project work. As my supervisor, he has constantly encouraged me to keep on focused on achieving the goal.His observations and comments helped me to establish an overall direction for the research and to make an in depth study.He has been my source of inspiration throughout the project work and without his invaluable advice and assistance it would not have been possible for us to complete this thesis. I am very much thankful to the faculty members of Department of Computer Science and Engineering, National Institute of Technology, Rourkela for their constant support and help during the project work. I would also like to thank my family and friends for their constant motivation which helped me in the successful completion of the work.

Subhendu Mishra

Abstract

Program slicing has been a hot topic for research nowadays because of its use in program debugging, code simplification and code analysis. As the maintenance of softwares takes up to 60% of programming effort and cost, its reduction has become vital. Certain blocks of code can cause unnecessary complexity which doesnt even contribute to the actual computations in the program. This project is concerned with construction of System Dependence Graph **(SDG)** for an input source code. SDG is an intermediate graphical representation to represent the dependencies among different programming constructs. The graph is constructed by converting the source code into tokens which are then analysed to form nodes. The nodes of the graph are connected by edges that represent different dependencies present in the program. A node is added to the graph when we encounter any variable or function. Edges are added to the graph on the basis of what relation the nodes have in between them. A control dependency edge is added between two nodes when one node follows the other in the execution sequence. A data dependency edge is added when the value of a certain variable is used at some other point of interest in the program. To compute the slices from the SDG, we have implemented the two phase algorithm as proposed by Horwitz. In this algorithm, for a given slicing criterion, we calculate the **phase 1** slice without descending into the called procedures. We mark all the nodes which are reachable from the given slicing criterion except for the ones which are related by the parameter-out edges. The **phase 2** slice is calculated by marking all the nodes in the graph except for those related by parameter-in edges and call edges. The slices from the two phases are then merged to give the final static backward slice for the entire source code.

Keywords: slice, node, dependency edge, System Dependence Graph, interprocedural.

Contents

List of Figures

Chapter 1

Introduction

When software is developed it is common to have situations in which software functionalities are no longer working due to certain changes being made in the program. The level of difficulty in finding the bug increases as software application grows in size and complexity. So with aging of software it gets harder to understand and maintain. Program Slicing[11]is a method for reducing programs by focusing on selected aspects of semantics. In this process we delete those parts of the program which have been determined to have no effect upon the semantics of interest. The reduced program, called as slice, is an independent program shall completely represent the original piece of program with the specified features. Therefore we preserve only those statements which have some effect with respect to the given slicing criterion. Slicing has various applications in testing and debugging, reverse engineering, code analysis, program differencing. Slicing focuses attention on those parts of the program which may actually contain a fault and thereby reducing our effort and time in finding a bug. There are various types of program slicing techniques some of which are static slicing, dynamic slicing, forward slicing, backward slicing, conditioned slicing etc. The input program which is to be sliced is represented by an intermediate graphical representation. There are various kinds of intermediate representation each depicting some different meaning. Some of the representations are Control Dependence Graph (CDG), Data Dependence Graph (DDG), Program Dependence Graph (PDG)[7], System Dependence Graph (SDG)[4] etc. These graphs are used for slicing purpose to find the minimal structure of the program and even to localise bugs to smaller part of the program.

1.1 Motivation

Software maintenance takes a huge part of time and effort of the total software development process .As the size of the software increases so does its complexity which increases the maintenance cost and time taken to for testing and debugging. The input database for the software is huge which makes it difficult to build a complete test suite which can analyse and fix the bugs. We can localise the errors as most of the code part does not contribute to the errors and concentrate on the localised regions. Program slicing methods can be used to localise such errors which will speed up the testing and debugging process.

1.2 Objective

The objective of this project is to build an intermediate graphical representation of the given program by statically analysing the code with the help of a lexer and parser. Then we apply the slicing algorithm on the input graph to find the static backward slice.

Chapter 2

Basic Concepts

2.1 Program Slicing

Slicing is a method of code analysis which is used to abstract a set of related statements in a code which bind together to perform a particular computation. The set of statements obtained is known as slice. It finds out the statements which affect the value of a variable at the point of interest in a particular program .Originally proposed by Mark Weiser as a method for reducing programs by analysing their data flow and control flow starting in the program. The program is reduced into a smaller form which can behave similar to that of original program with respect to the point of interest. The input to the slicing algorithm is usually an intermediate graphical representation of the given program that is to be sliced, and the output is program slice [1] which is found out with respect to the given point of interest.

2.1.1 Slicing Criterion

The slicing criterion <X, V> is defined such that we can find a particular slice. The X is the statement in which the variable is present and V is the variable for which the slice is to be found out.

2.1.2 Different types of slicing

2.1.2.1 Forward Slicing

In this type of slicing we show which all statements are affected in the program due to the given slicing criterion <X, V>. The affected statements form the forward slice.

2.1.2.2 Backward Slicing

In this type of slicing we show those statements which can affect the given slicing criterion <X, V>. The above setoff statements form the backward slice.

2.1.2.3 Static Slicing

This type of slicing is done for the source code for all possible set of input values. The values are predefined and the slicing is done without actual execution of the code. The slices may become big in size if the code size is very large. It contains all the possible statements which may affect the variable.

2.1.2.4 Dynamic Slicing

The slices in this type of slicing are derived from the particular execution of a program. The slices are computed with respect to program history. The slices produced are relatively small and contain all the statements that actually affect the value of a variable.

2.1.3 Various Types of Dependencies

2.1.3.1 Control Dependency

A statement is said to be control dependent on a preceding statement if the preceding statement decides whether its following statement should be executed or not. This only happens when the statement is in branch of a preceding statement and is controlled by a condition.

2.1.3.2 Data Dependency

A statement is said to be data dependent on another statement when it refers to the data of the other statement. For example if a variable is defined in a statement a and it is used in statement b, then b is data dependent on a.

For a sample program we show both type of dependencies below

The sample program

```
int x=0;
if (x==0)
{
x=x+1;
int y=10;
}
else
{
printf ("Hello");
x=2;
}
int z=y+2;
```

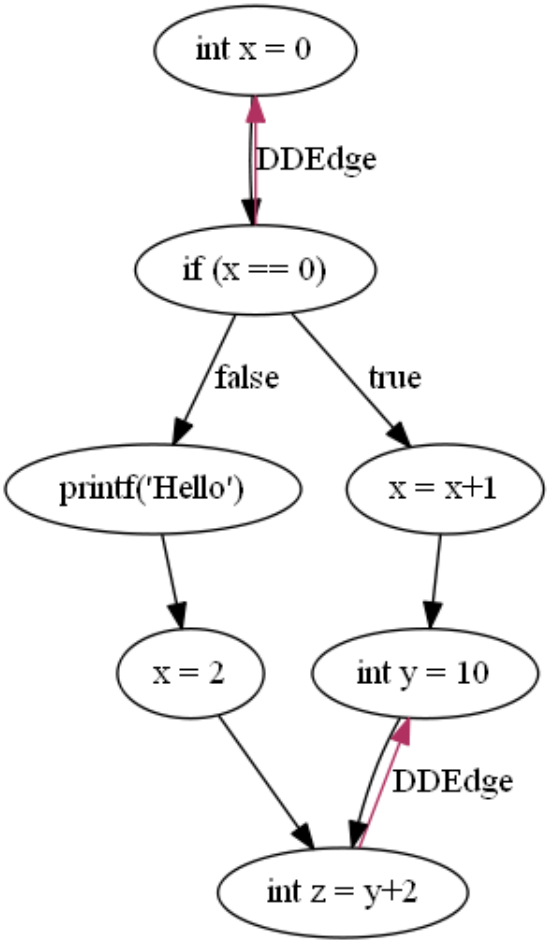

FIGURE 2.1: Control Dependencies and Data Dependencies

The red edge is denoted as data dependency edge and the black edges are denoted as control edges. Here "if (x==0)" is data dependent on "int x=0" and "int z=y+2" is data dependent on "int y=10" "printf('Hello')" and "x=x+1" are control dependent on "if (x==0)".

2.1.4 Different types of Intermediate Graphical representations

2.1.4.1 Control Flow Graph

A Control Flow Graph (CFG) is a graphic representation of a program and its flow of execution. It represents all possible set of statements of a program. They consist of all the typical building blocks of any flow diagrams. There is always an Entry node, and an Exit node with the flows (or arcs) between nodes. Nodes are various statements in the program code and are labelled accordingly.

A sample program with its CFG

1: main()
2: {
3: m=4;
4: i=4;
5: x=4;
6: while(i¡=m){
7: x=x+9;
8: write (x);
9: i=i+1;
10: }
11:

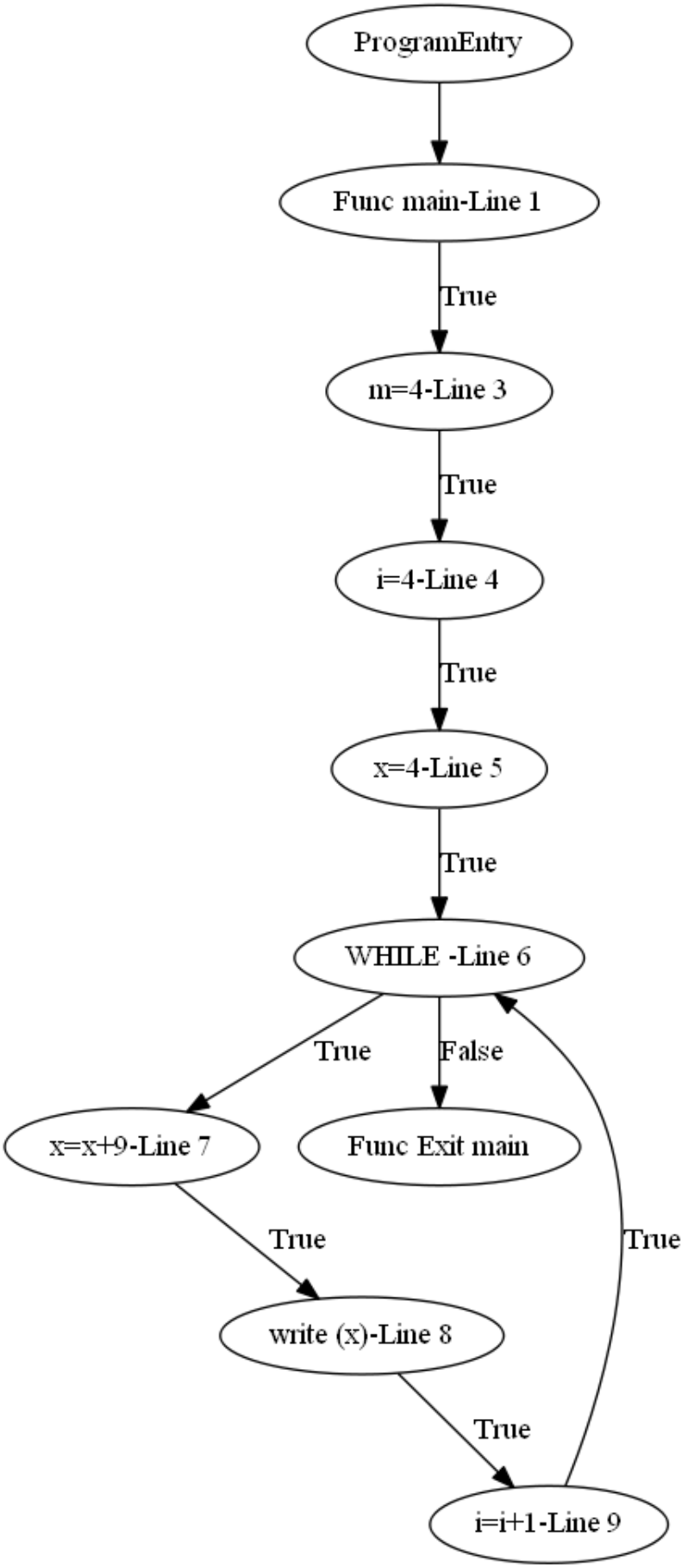

FIGURE 2.2: CFG of sample program

2.1.4.2 Program Dependence Graph(PDG)

[7] The control flow graph shown together with data dependencies is known as program dependency graph (PDG). It has the same properties as that of control flow graph. It has an entry node and exit node with edges representing the data dependencies and the control flow among the nodes

For the above given sample program its respective PDG will be as follows

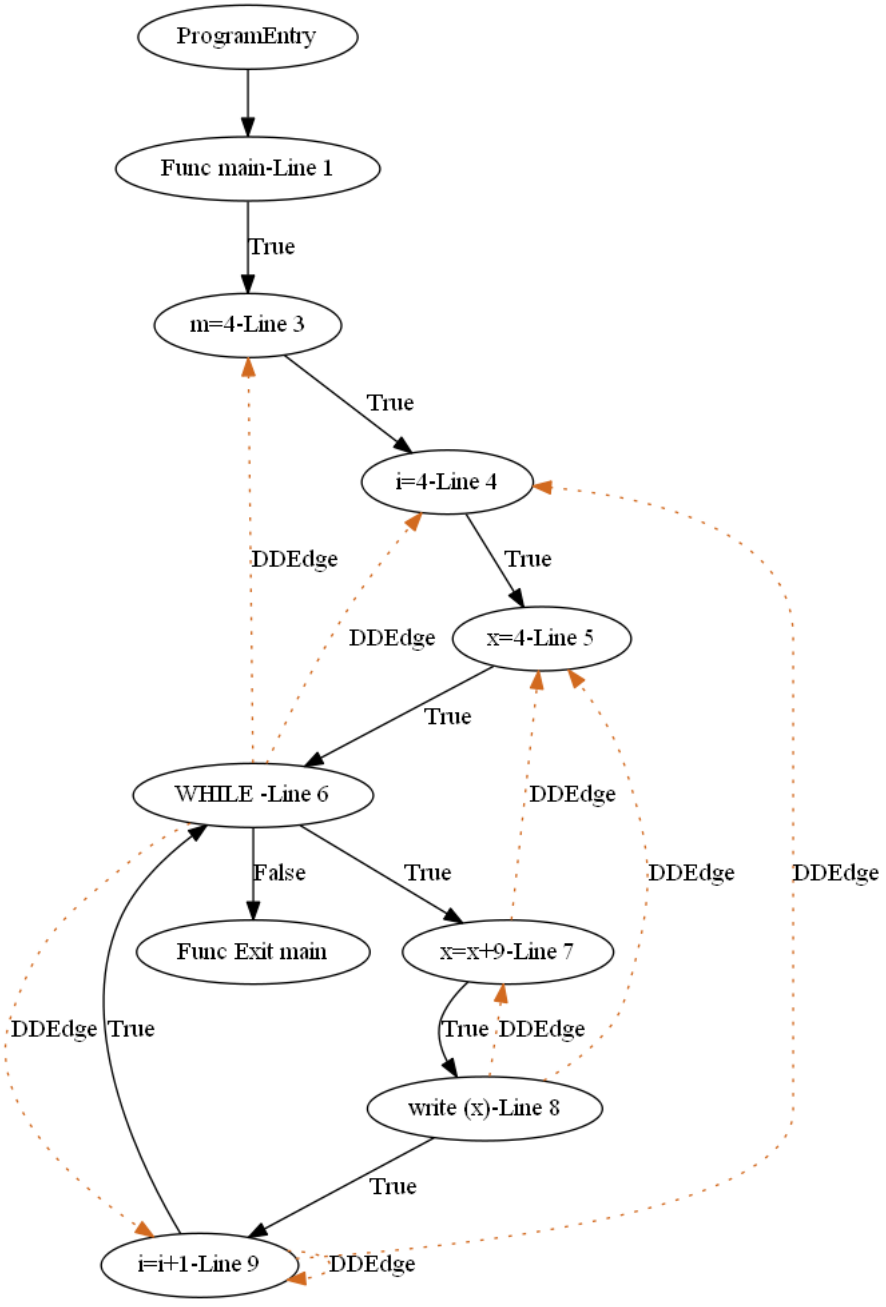

FIGURE 2.3: PDG of sample program

2.1.4.3 System Dependence Graph

[4] When we combine all PDGs with the transitive dependencies among the procedures, it gives us a model of inter-procedural dependencies which is known as System Dependence Graph (SDG).It was originally proposed by Horwitz et. al. Various procedures with their dependencies i.e. control and data dependencies with call edges parameter edges and summary edges forms the graph. It shows all the characteristics of a PDG individually on the graph. There is an entry and exit node for each procedure. When there is a call made to another procedure at the call site actual parameters are associated with the called procedure The number of actual parameters are equal to the number of arguments passed. Also a call edge is shown from the call site to the entry node of the called procedure. At the entry node of the called procedure formal parameters are associated with the procedure .The formal parameters are then associated with their corresponding actual parameters The actual-in parameters are associated with their corresponding formal-in parameters and actual-out parameters are associated with their corresponding formal-out parameters. Summary edges are added to show the transitive flow of data which happen when procedure calls are made. For another sample program we show its corresponding SDG

sample program

```
int main()
{
int a = 10, b;
b = add(a,3);
return 0;}
int add(int x, int y) {
x = x + y;
return x;
}
```

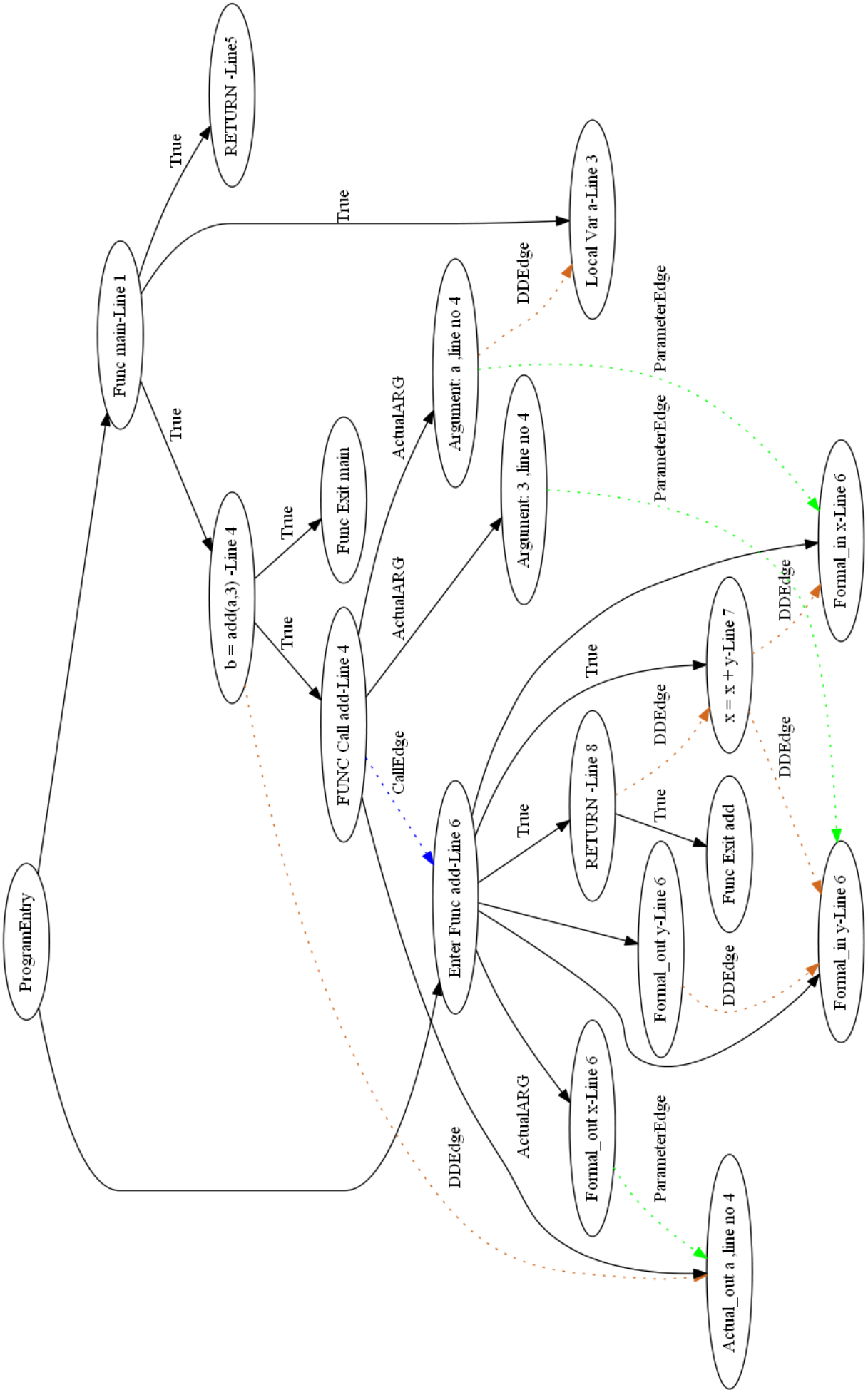

FIGURE 2.4: SDG of the sample program

Chapter 3

Literature Review

The idea of program slicing was originally introduced by M. Weiser[11] where we calculated the slices for flow graphs and hammock graphs. Here we took the union of transitive closures of dependencies in the callers of procedure P represented by **UP(C)**.and dependencies in the called procedure in P represented by **DOWN(C)**. to give the final slice .But it did not give accurate slices. Gallagher proposed a modification to the Weisers algorithm. In this algorithm, goto statements are included in the slice. The algorithm does not produce correct slices in all cases. Then Ferrante et. al.[7] proposed an intermediate graphical representation of a program, known as Program Dependence Graph (PDG) In this approach we show control and data dependencies for each operation in the program .Control flows are shown to visualise the flow relationship between the various statements in the program. Various optimizations can be used as only computationally related statements are connected together But it could be used only for intraprocedural slicing.. Ball and Horwitz [8] and Choi and Ferrante [9] discovered themeselves that previous PDG-based slicing algorithms produce inefficient results in the presence of unstructured control flow, slices may compute values at the criterion that differ from what the original program does. These problems happen because the algorithms cannot determine correctly whether unconditional jumps such as break, goto, and continue statements are required in a particular slice. They proposed two similar algorithms to determine the relevant jump statements to include in a slice. Both of them require that the jumps be represented as pseudo-predicates and the control dependence graph of a program be constructed from an augmented flow-graph of the program and two formal proofs have been proposed to show their algorithms compute correct slices. To overcome the limitation of PDG, Horwitz

et. al.[4] proposed System Dependence Graph(SDG).In this approach we combine the dependencies of various procedures together and all the procedures are related to each other by various other parameters They used a two phase algorithm to obtain the slice from the representation. The slicing is done with respect to a particular procedure as well as taking into account procedure calls and other transitive dependencies happening due to these procedure calls. Korel and Laski[1] introduced the concept of dynamic slicing . It was different from static slice as a dynamic slice is constructed with respect to only a particular input of a program which shall give a particular output It does not include the statements that have no relevance with the slicing criteria on some particular input. Because of the facilities like the run-time handling of arrays and pointer variables, dynamic slicing algorithms were found out to be easier compared to static algorithms and also result are more precise. It was quite helpful for debugging purpose. Agrawal, Horgan [13] extended this approach to relevant slicing. A relevant slice with respect to a variable contains not only the statements that have an influence on the variable but also those executed statements that did not affect the output, but could have affected it had they evaluated differently. Relevant slicing can facilitate incremental regression testing.. But it was only for procedural programming. Livadas . al.[6] proposed a simpler method of computing the summary edges. by deriving the information from the construction of the system dependence graph. They constructed the SDG in a bottom up fashion and it require only one copy of each PDG.Also Larsen and Harrold[12] improvised the system dependence graph for the representation of object-oriented programs and used the two phase algorithm of Howritz et. al. with simple modifications to compute static slices.
The work done is basically till now was going on for static analysis of sequential programs i.e. programs are analysed with actual execution of programs and a lot of development is going on for object oriented programs But now the industries have software which have hierarchical dependencies and are quite distributed in nature. We have real time systems which need to pass the data within the deadline. Development is still required in these fields.

Chapter 4

Proposed Work

- We first need to construct an intermediate graphical representation of the sample input program which is the System Dependency Graph .

- Then we implement the two-phase algorithm as proposed by Horwitz et. al.[4] to obtain the static slice for the given program.with respect to the slicing criterion.

4.1 Constructing the System Dependence Graph

The graph is built by taking an input source file which is read and with the help of lexical and syntactic analysis its parse tree is built and when it is associated with data dependencies and control dependencies with inter procedural dependencies we get its graph representation
The approach for the construction is given as follows

Requirement Source Program
Result SDG of the Source Program
Open file("path of source file")
while EOF not encountered **do**
Put line in array **end while**
for each line in array**do:**
search for main
if main found **then**
Store index
BREAK
else
CONTINUE
end if
end for
Find_mainspan (return lines between "{" "}").
while "}" not encountered **do**
Search_function(return function_name, parameters passed)
end while
for each function_name
Find function_definition in whole program and return lines in function_definition
end for Connect function_call to function_definition by comparing function names by Call Edges.
for each function_call **do**
for each parameter passed **do**
Establish actual_in parameters and actual_out parameters
end for
end for
for each function_definition **do**
Establish formal_in paramters and formal_out parameters at function_defintion

Store variable definition if found
Store assignments of variables if found
Store loop condition if found
Store conditional statements if found
Relate all tokens with control flow and data dependencies
end for
Connect the formal_in parameters with corresponding actual_ in paramters and formal_out parameters with actual_out parameters by parameter edges.

4.2 Constructing the graph

For the construction of graph we require the various function definitions and function usages with the control flow and data dependencies and inter procedural dependencies which we get from the above algorithm. The graph file created has got a .graph extension and is created by file handling. It can be viewed by Graphviz software.
A sample generated graph file is given as follows
digraph CLDG{
node [color=lightblue2 , style=filled]
"Func main-Line 1" → "Local Var a-Line 3" [label="True"];
"Func main-Line 1" → "Local Var b-Line 3" [label="True"];
"Local Var b-Line 3" → "EXP 5-Line 4" [label="True"];
"EXP 5-Line 4" → "FUNC Call add-Line 4" [label="True"];
"FUNC Call add-Line 4" → "Enter Func add-Line 6"
[style=dotted,color=blue,label="CallEdge"];
Local Var a-Line 3" → "Argument: a ,line no 4"[style=dotted,color=chocolate,label="DDEdge"]
"FUNC Call add-Line 4" → "Argument: a ,line no 4" [label="ActualARG"];
"FUNC Call add-Line 4" → "Argument: 3 ,line no 4" [label="ActualARG"];
"ProgramEntry" → "Func main-Line 1";
"ProgramEntry" → "Enter Func add-Line 6";
"Enter Func add-Line 6" → "Formal_in x-Line 6";
"Enter Func add-Line 6" → "Formal_in y-Line 6";
"Argument:a ,line no 4" → "Formal_in x-Line 6"
[style=dotted,color=green,label="ParameterEdge"];
"Argument:3 ,line no 4" → "Formal_in y-Line 6"

[style=dotted,color=green,label="ParameterEdge"];
"Formal_out x-Line 6" → "Actual_out a ,line no 4 "
[style=dotted,color=green,label="ParameterEdge"];
"Formal_in x-Line 6" → "EXP 13-Line 7" [style=dotted ,label="DDEdge"];
"RETURN -Line 8" → "Formal_out x-Line 6" [style=dotted, label="DDEdge"];
"Formal_in y-Line 6" → "EXP 13-Line 7" [style=dotted ,label="DDEdge"];

4.3 The Two-Phase Algorithm

The algorithm has two phases as by the name is suggested has two phases. For a particular slicing criterion <x ,v> which is present in some procedure P we do the following:
The Phase-1 We find out the vertices which can reach x which are either present in the same procedure P or any procedure that calls P. While finding out the vertices we follow only the parameter-in edges and call edges while traversing from one procedure to another
The Phase-2 We find out the vertices which can reach x which are present in the same procedure P or any of the procedures which are called by P. Here we only follow the parameter out edges.

The algorithm can be represented in a pseudo code as follows:
Requirement SDG in .graph format
Result Phase-1 Slice, Phase-2 Slice and Final Slice w.r.t Slicing Criterion
Open file("Path of .graph file")
Enter slicing_criterion
for each line in file **do**
if slicing_criterion found **then**
compute slices
end if
end for
Calculate_slice_1(slicing_criterion)
for each line in original file **do**
Find function cotaining slicing_criterion
if function associated with parameter_out edges **then**
remove_parameter_out (delete parameter_out edges)

end if
if function associated with call edge and parameter_in edges **then**
Traverse along those edges which can be used to reach slicing_criterion
Write in output file (Traversed edges)
end if
end for
Calculate_slice_2(slicing criterion)
for each line in original file **do**
Find function containing slicing_criterion
if function associated with call edge and parameter_in edges **then**
remove_transitive (delete parameter_in edges and call edges)
end if
if function associated with parameter_out edges **then**
Traverse along those edges which reach slicing_criterion
Write in output file (Traversed edges)
end if
end for

Chapter 5

Implementation and Results

Here we present the various implementation and the results which include various screenshots of the construction of the intermediate representation of the input program and slicing algorithm to compute the static backward slice of a statement. The implementation was done by taking only C programs into account which represent similar to that of interprocedural programs.

5.1 Tools used

We use the following tools in order to implement and code the programs and finally to get the intermediate representation.

- Eclipse
- ANTLR [15]
- Graphviz

5.1.1 Eclipse

It is a multi-language software development environment constituting of an integrated development environment and an extensible plug-in system. It is written primarily in Java and can be used to develop applications in Java and, but it can

be used for other languages by means of the various plug-ins. The other languages include C, C++, COBOL, Python, Perl, PHP, and others. The IDE is often called Eclipse ADT for Ada, Eclipse CDT for C, Eclipse JDT for Java and Eclipse PDT for PHP. The most important feature of Eclipse is its plug-in system. We can integrate different plug-in tools into the eclipse environment and can be used them in the applications. It is also simple to use as we need not install it. The only thing that we have to do it is to download Eclipse and run the eclipse.exe file. .We have to download and install MinGW GCC compiler for compilation of C++ code .

5.1.2 ANTLR

[15]

ANTLR stands for ANother Tool for Language Recognition. It is a top-down parser generator that uses LL (*) parsing. ANTLR takes as input a grammar that specifies a language and generates as output, source code for a recognizer for that language. ANTLR supports code in C, Java, Python, and C. It provides a single consistent notation for specifying lexers, parsers and tree parsers. This is in contrast with other parser/lexer generators and adds greatly to the tool's ease of use. It consists of a Lexer and a parser which is used to compile a input grammar file which contains definitions and rules for a certain language for the input programs to be read.

5.1.3 Graphviz

Graphviz is a tool that can pictorially represent a graph. We have used this tool in our project to visualize our final output in a better way i.e. in the form of a pictorial graph instead of a adjacency matrix or adjacency list. The output of the program is converted into a form that is recognizable by graphviz and is written into an output file in the same format. Graphviz reads from the output file in order to generate the graph that the user can visualize.

5.2 Screenshots

The input sample program for which the SDG is made and following it the slicing algorithm was applied is given as below.

```
int find(int x) {
int q=0;
q= x+2;
printf("%d",q);
return q;
}
int main()
{
int x=20;
int z;
z= find(x)+1;
```

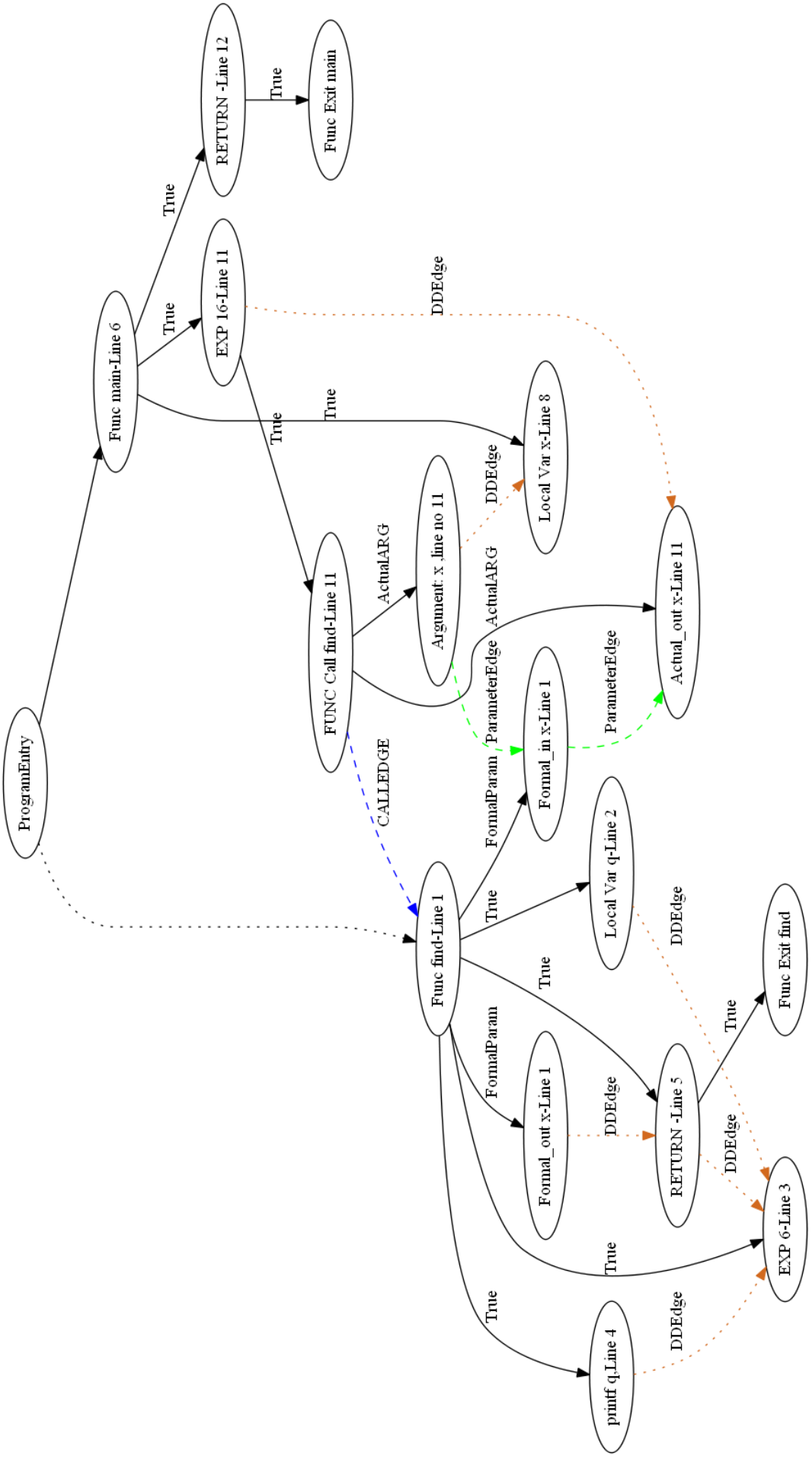

FIGURE 5.1: SDG of the source program

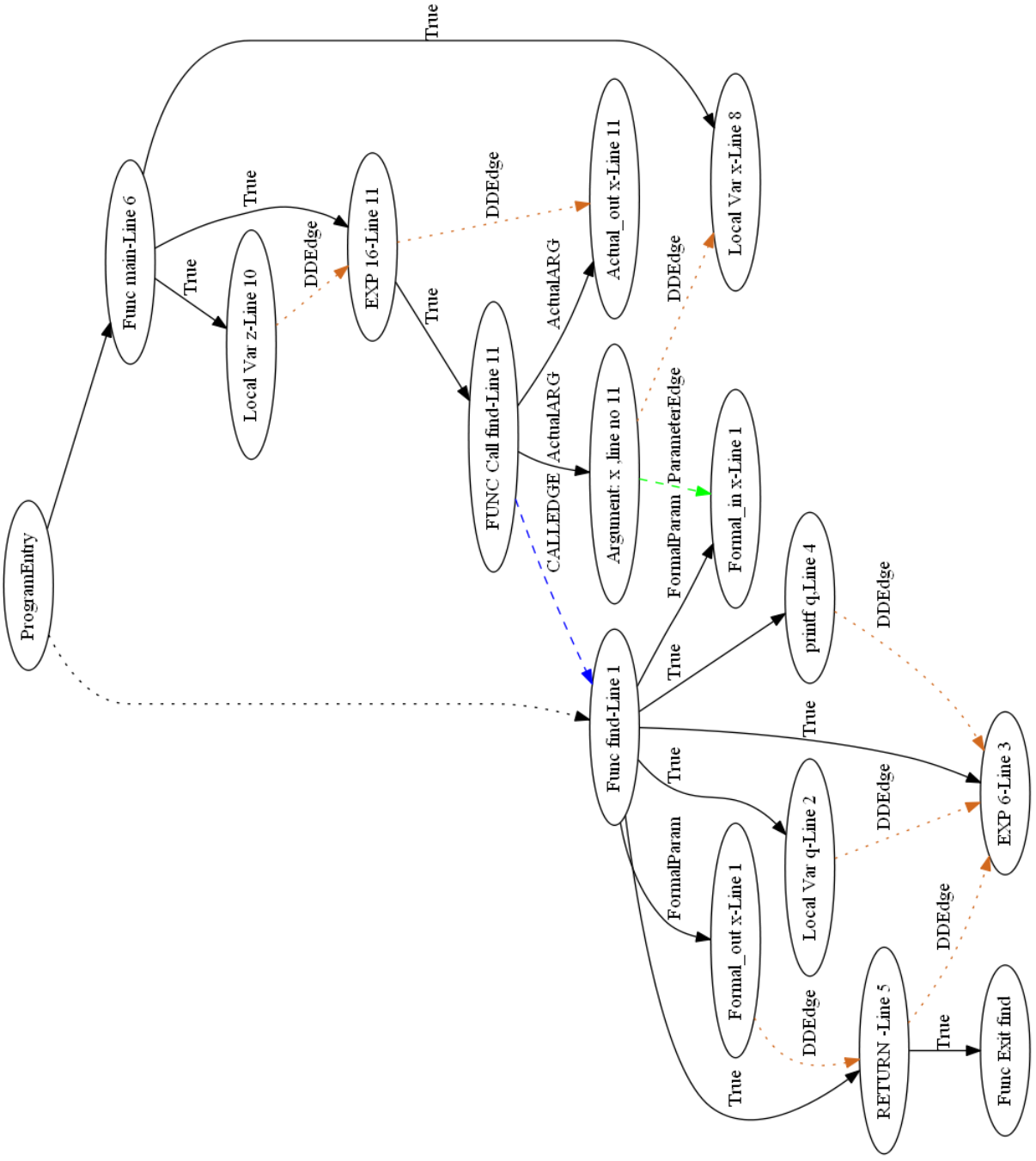

FIGURE 5.2: Slice obtained for slicing criterion <EXP 6,Line 3> for the program at phase 1

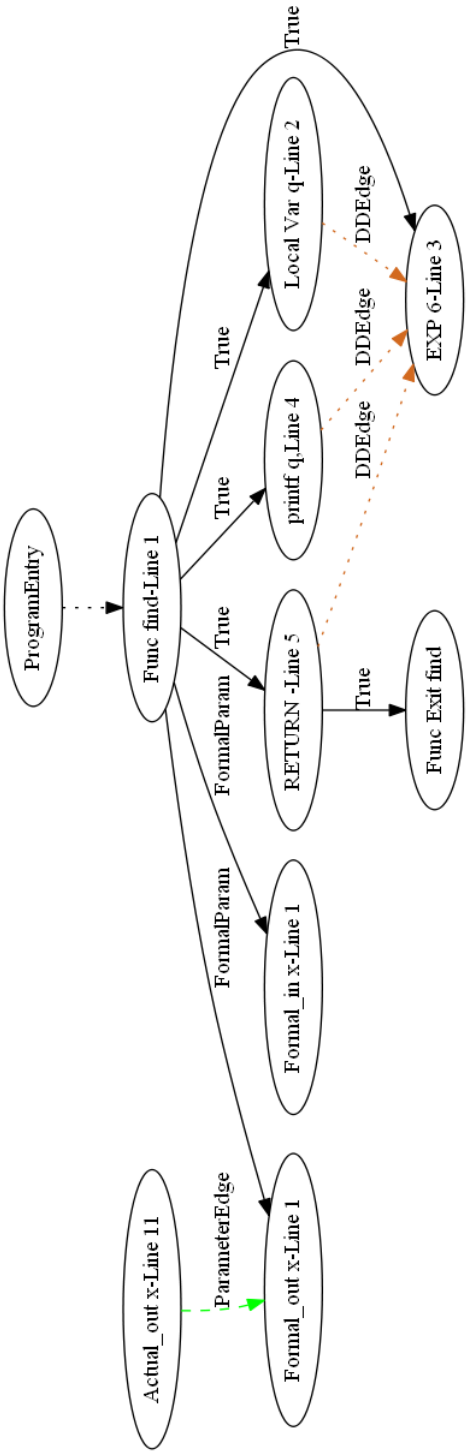

FIGURE 5.3: Slice obtained for slicing criterion <EXP 6,Line 3> for the program at phase 2

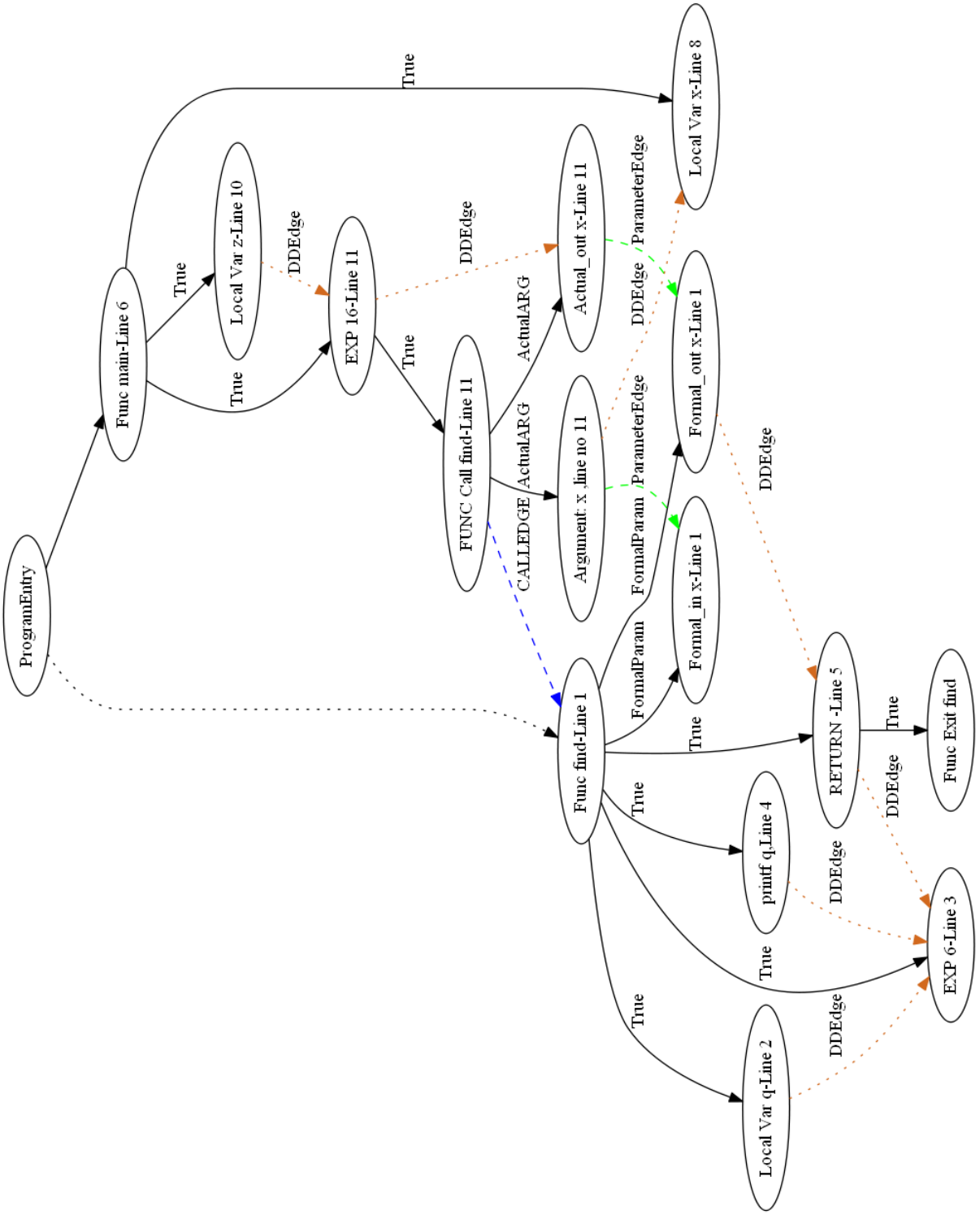

FIGURE 5.4: Final Slice obtained for slicing criterion <EXP 6,Line 3> for the program

5.3 Analysis of Algorithm

We made a table of number of lines and number of function calls made in different programs which we tested with the Two-phase algorithm with respect to the total time for building them.

No. of Lines	No. Of Function Calls	Time for Build(ms)
10	2	46
13	2	47
13	3	49
15	3	48
22	4	55
19	4	57.1
70	6	88.01
120	7	103.5

FIGURE 5.5: Table depicting No.of lines and No. of function calls vs.Build time

We can see from the table that the time to build increases with both the lines of code and number of function calls but relatively if same number of function calls are made in lesser number of lines of code it takes more time.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We have been able to develop an intermediate representation known as System Dependence Graph for any input program. After generating the SDG, we have implemented the two phase slicing algorithm to calculate slices of the program based on provided slicing criterion. The slices show the same behaviour as that of original program with respect to the slicing criterion which should help the programmers to a great extent in debugging by localising the errors to certain particular areas. We also analysed the algorithm by making a table between time taken to build slices with its intermediate representation versus number of lines of code used in the program. The results show that The time to build increases with both the lines of code and number of function calls but relatively if same number of function calls are made in lesser number of lines of code it takes more time.

6.2 Future Work

The approach was used for programs which replicated interprocedural programs.for calculation of static slices This approach can be extended to the calculation of dynamic slices of such programs and then can be used for object oriented programs It also can be extended for software testing purposes for selecting the optimal test suite and it can also help in the debugging process by localising the bugs.

Bibliography

[1] B. Korel and J. Laski, "STAD - a system for testing and debugging: user perspective", *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, pages 13-20, July 1988.

[2] G. B. Mund, R. Mall, S. Sarkar, "Computation of intraprocedural dynamic program slices", *Information and Software Technology*, Vol. 45, pages 499-512, 2003.

[3] D. Binkley, K. Gallagher, "Program Slicing", *Advances in Computers, Academic Press*, Vol. 43, pages 1-50, 1996.

[4] S. Horwitz , T. Reps, D. Binkley , "Interprocedural slicing using dependence graphs", *ACM Transactions on Programming Languages and Systems*, pages 26-60, January 1990.

[5] G. B. Mund, R. Mall, "An efficient interprocedural dynamic slicing method",*The Journal of Systems and Software* ,Vol. 79, pages 791-806, 2006.

[6] P. E. Livadas, T. Johnson, "An optimal algorithm for the construction of the system dependence graph",*Information Sciences*, Vol.125, pages 99-131, 2000.

[7] J. Ferrante, K. J. Ottenstein , J. D. Warren, " The Program Dependency Graph and its Use in Optimization",*ACM Transactions on Programming Languages and System* , Vol. 9, No. 3, pages 319-349,1987.

[8] T. Ball and S. Horwitz, "Slicing program with arbitrary control-flow", *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, P. Fritzson, Ed., vol. 749 of Lecture Notes in Computer Science, Springer-Verlag, pages 206-222, 1993.

[9] J. Choi and J. Ferrante, "Static slicing in the presence of goto statements",*ACM Transactions on Programming Languages and Systems* , Vo1. 6(4), pages 1097-1113, July 1994.

[10] M. Weiser, "Programmers use slices when debugging",*Communications of the ACM*,Vol. 25 (7), pages 446-452, 1982.

[11] M. Weiser, "Program Slicing",*IEEE Transactions on Software Engineering*, VOL. SE-10, NO. 4, July 1984.

[12] L. Larsen and M. J. Harrold, "Slicing Ob-ject-Oriented software",*International Conference on Software Engineering Proceedings of the 18th international conference on Software engineering* , Berlin, Germany, pages 495-505, 1996.

[13] H. Agrawal and J. R. Horgan, "Dynamic program slicing",*Proceedings of the ACM SIGPLAN90 Conference on Programming Language Design and Implementation* , pages 246-256, June 1990

[14] M. Kamkar, "An overview and comparative classification of static and dynamic program slicing", *Technical Report LiTH-IDA-R-91-19*, Linkoping University, Linkoping, 1991.

[15] T. J. Parr, R. W. Quong,"ANTLR: A Predicated-LL(k) Parser Generator" ,*Software Pratice and Experience*, VOL. 25(7),pages 789810,July 1995.

[16] K.B. Gallagher and J.R. Lyle," Using program slicing in software maintenance", *IEEE Transactions on Software Engineering*, Vol. 17(8), pages 751-761, 1991.